ACM@IITR

AutoJudge: Predicting Programming Problem Difficulty

Architkumar J. Modi

Project Overview

Online competitive programming platforms such as Codeforces, CodeChef, and Kattis assign difficulty labels (Easy / Medium / Hard) and numerical difficulty scores to problems. These labels are typically determined through human judgment and community feedback.

AutoJudge is a machine learning–based system that automatically predicts:

- **Problem Difficulty Class** → Easy / Medium / Hard (Classification)
- **Problem Difficulty Score** → Numerical value (Regression)

The prediction is made using only the textual content of the problem, without any user statistics or historical submissions. The project also includes a simple web interface that allows users to paste a problem statement and instantly view predicted difficulty.

Dataset Used

The dataset (`problems_data.jsonl`) consists of programming problems collected from online judges. Each data sample includes the following fields:

- `title`
- `description`
- `input_description`
- `output_description`
- `problem_class` (Easy / Medium / Hard)
- `problem_score` (numerical difficulty score)
- `sample_io` (not used for modeling)
- `url` (not used for modeling)

Only textual fields (`description`, `input_description`, `output_description`) are used for feature extraction, as required by the project constraints.

Data Preprocessing

Firstly, it was checked whether data contained any missing values or not. No missing values were there:

```
df.isnull().sum()
```

```
]: title                0
   description          0
   input_description    0
   output_description   0
   sample_io            0
   problem_class        0
   problem_score        0
   url                  0
   dtype: int64
```

Then the following fields were combined into a single textual input:

- Problem description
- Input description
- Output description

This ensured that the full problem context was captured:

🛠️ Step 2: Combine important text fields into one single text feature ¶

```
[2]: df['combined_text'] = df['description'] + " " + df['input_description'] + " " + df['output_description']

# Drop useless columns which can add noise
df.drop(['title', 'description', 'sample_io', 'url'], axis=1, inplace=True)

# Encoding classes in increasing order of difficulty:
df['problem_class'] = df['problem_class'].map({
    'easy': 0,
    'medium': 1,
    'hard': 2
})
df.head()
```

Text Preprocessing

Text preprocessing was carefully designed to avoid feature leakage and double counting:

- Unicode normalization
- Removal of LaTeX expressions and subscripts
- Removal of punctuation and formatting artifacts
- Stopword removal
- Lemmatization using spaCy
- Lowercasing and whitespace normalization

Feature Engineering

To comprehensively capture both the semantic meaning and the inherent structural complexity of the programming problems, a multi-faceted feature engineering approach was employed, incorporating three distinct feature types:

a) TF-IDF Features (Term Frequency-Inverse Document Frequency)

- **Application:** Applied to the pre-processed (cleaned) problem text.
- **Filtering:** Vocabulary was strictly limited to the top 2000 features to reduce noise and maintain computational efficiency. Rare and excessively common words (which lack discriminative power) were filtered out.

b) Keyword-Based Features

- **Purpose:** Quantify the presence of specific algorithmic concepts.
- **Concepts:** Weighted counts were calculated for crucial algorithmic concepts, including: `dp`, `graph`, `greedy`, `flow`, `binary search`, `dfs` and `bfs`.
- **Processing:**
    - Keywords were counted and converted into corresponding numeric features.
    - They were also *removed* from text prior to TF-IDF to prevent double counting.

Define weights (as per difficulty) of important keywords of programming problems:

[6]:
```
keyword_weights = {
    "dp": 2.5,
    "graph": 2.0,
    "greedy": 1.8,
    "math": 1.5,
    "geometry": 1.8,
    "string": 1.2,
    "shortest path": 3.0,
    "flow": 3.0,
    "matching": 3.0,
    "number theory": 2.5,
    "binary search": 2.0,
    "recursion": 1.5,
    "tree": 1.5,
    "dfs": 1.5,
    "bfs": 1.5,
    "combinatoric": 2.0,
    "modulo": 1.3,
    "xor": 1.3,
    "query": 1.2
}
```

c) Structural Features

- **Purpose:** Measure the innate complexity and formatting of the problem statement.
- **Features Included:**
    - **Mathematical Symbol Count:** The total number of mathematical symbols present (+, -, *, / , <, >, =, <=, >=). A higher count often indicates a more formal or mathematical requirement.
    - **Text Length:** The total number of tokens (words/punctuation) in the cleaned problem text.

Unified Preprocessing Pipeline

All preprocessing and feature extraction steps were implemented inside a single `sklearn` Pipeline using a `ColumnTransformer`.

This ensured:

- Identical preprocessing during training and inference.
- Robust integration with the web interface.

Model Selection and Training

Given the two distinct objectives—classification and regression—separate machine learning models were trained.

Classification (Easy / Medium / Hard)

The following models were evaluated (Hyperparameters were tuned) using cross-validation:

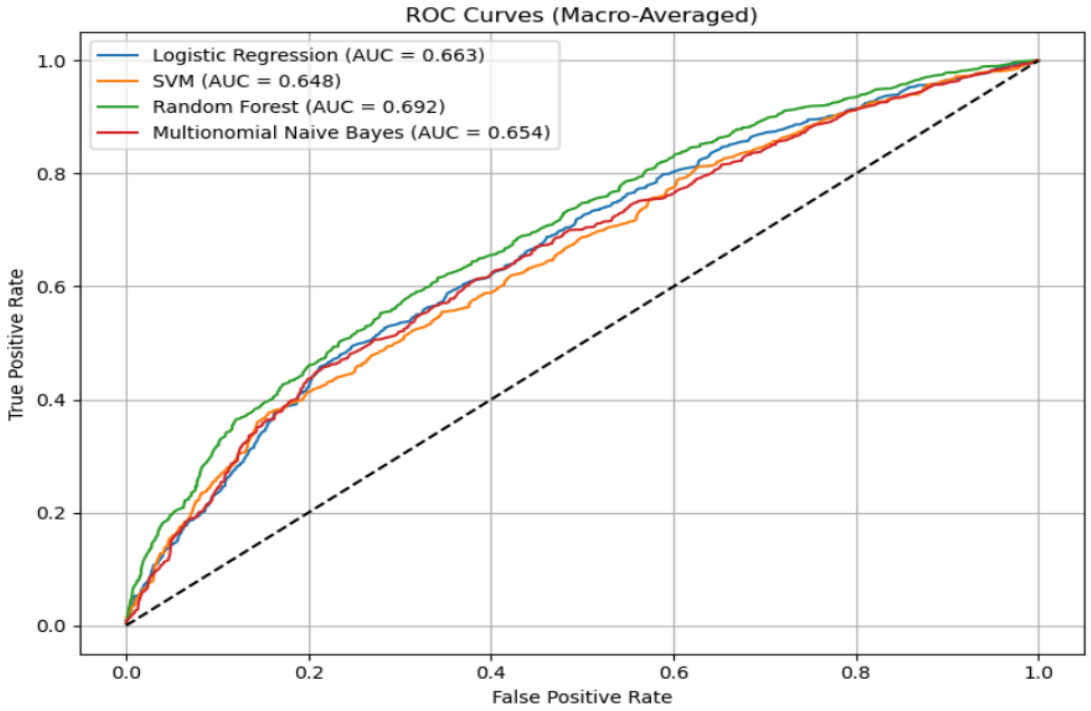
- Logistic Regression
- Support Vector Machine (SVM)
- Random Forest
- Multinomial Naive Bayes

Classification Results

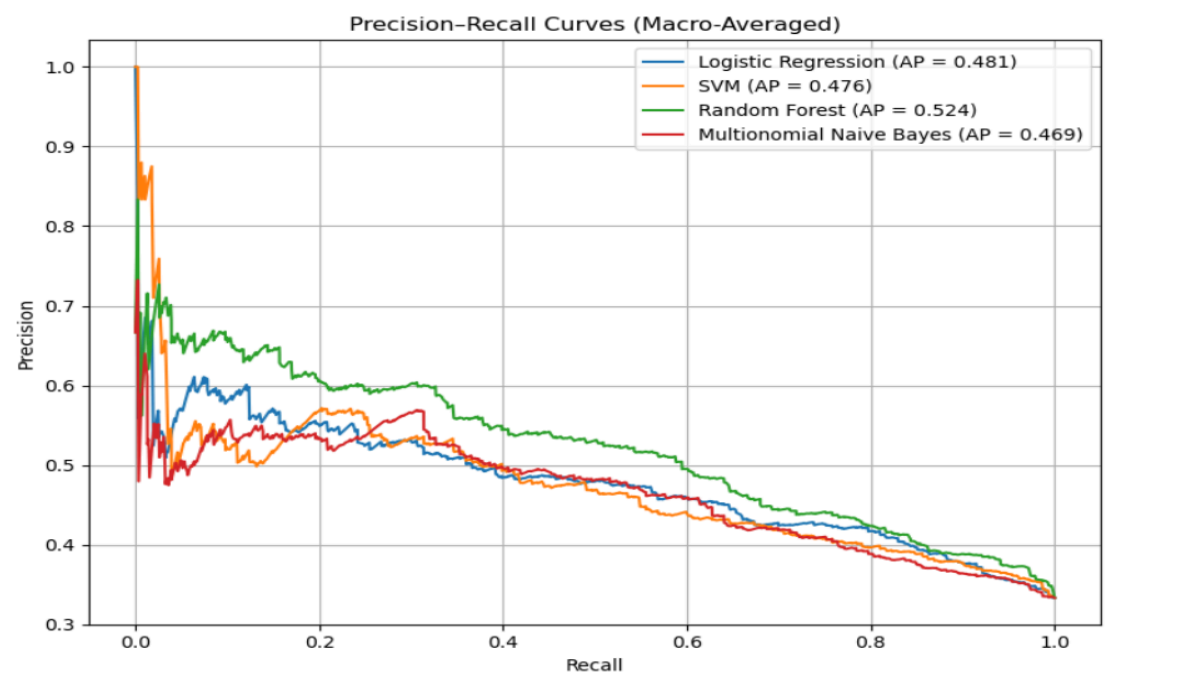
Model	Accuracy
Random Forest	0.537060
Multinomial Naive Bayes	0.506683
Logistic Regression	0.505468
SVM	0.496962

The **Random Forest** classifier demonstrated the highest accuracy, successfully predicting the difficulty class (Easy/Medium/Hard) in **53.7%** of the cases. This model was chosen as the final classification predictor and saved as "best_classification_model.pkl".

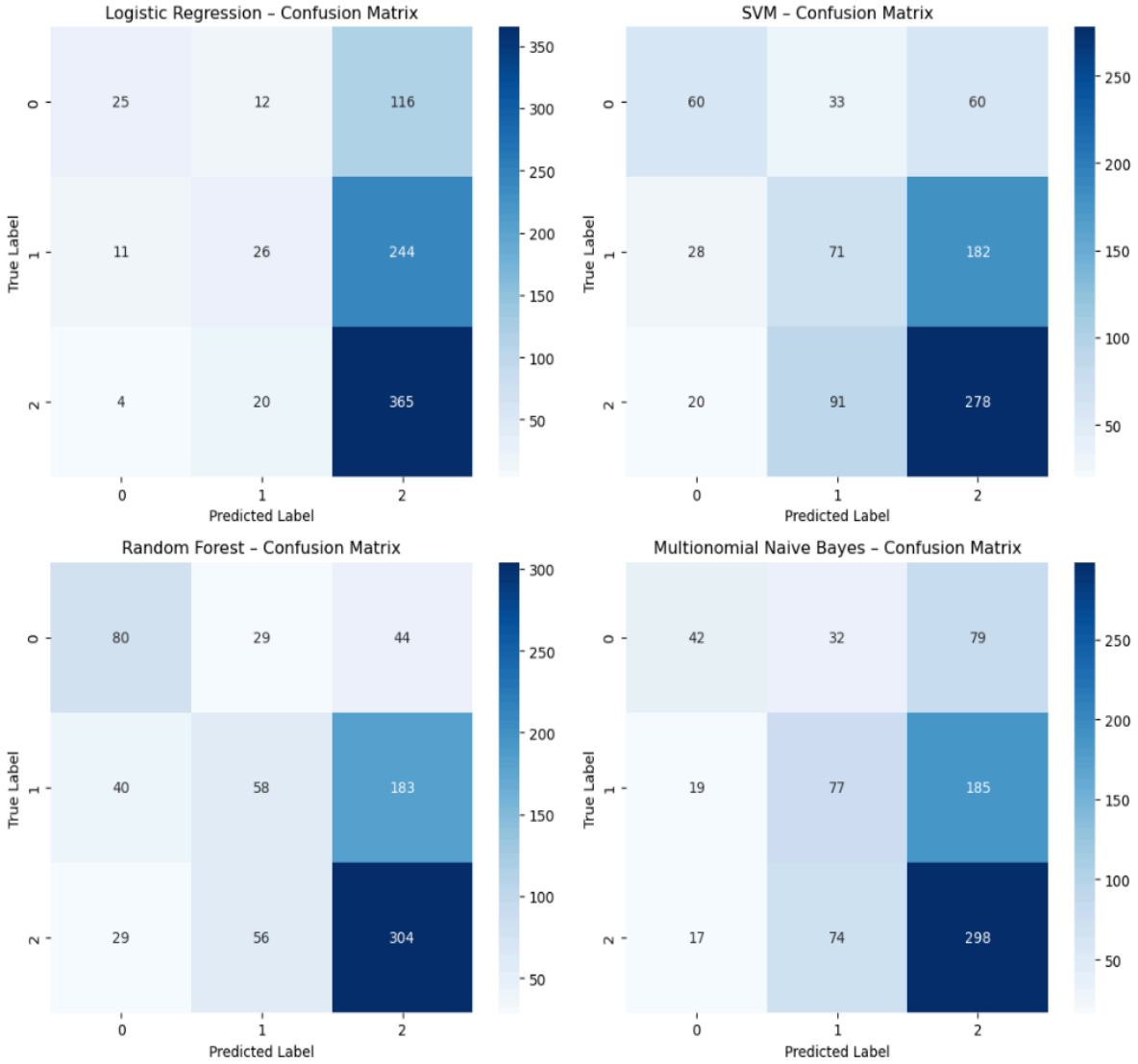
ROC Curves



Precision-Recall Curves



Confusion Matrices



Regression (Numerical Score)

The following models were evaluated using cross-validation:

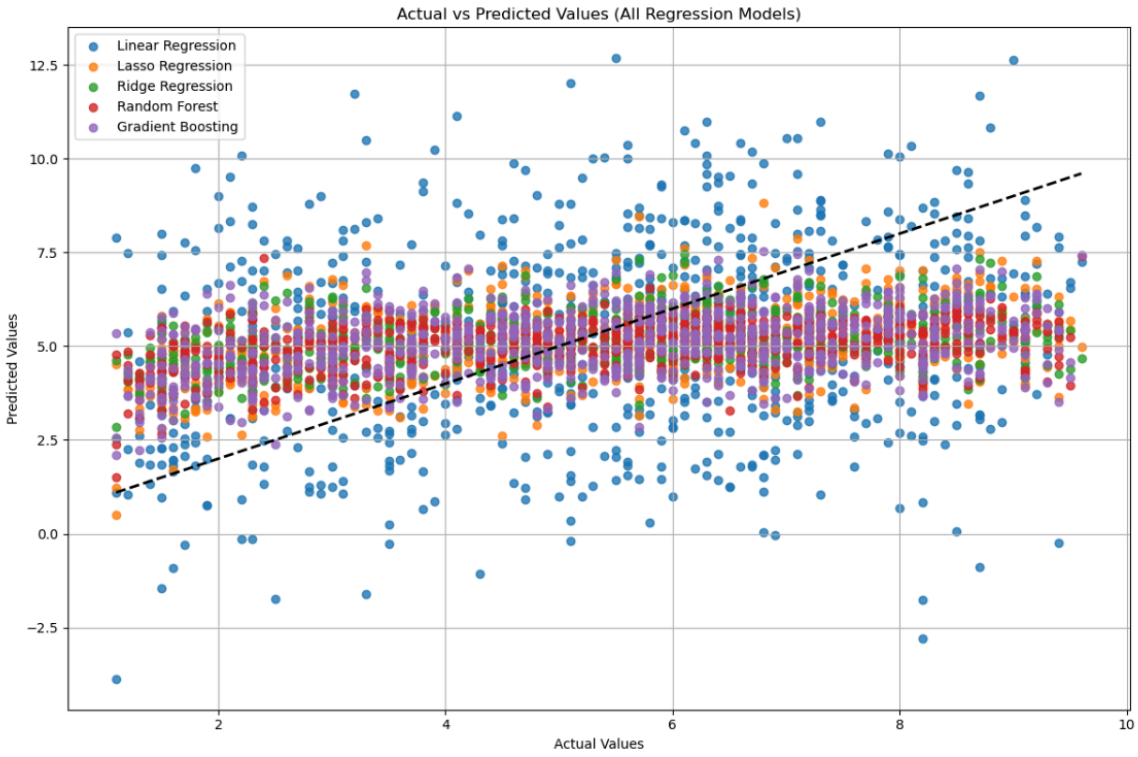
- Linear Regression
- Ridge Regression
- Lasso Regression
- Random Forest Regressor
- Gradient Boosting Regressor

Regression Results

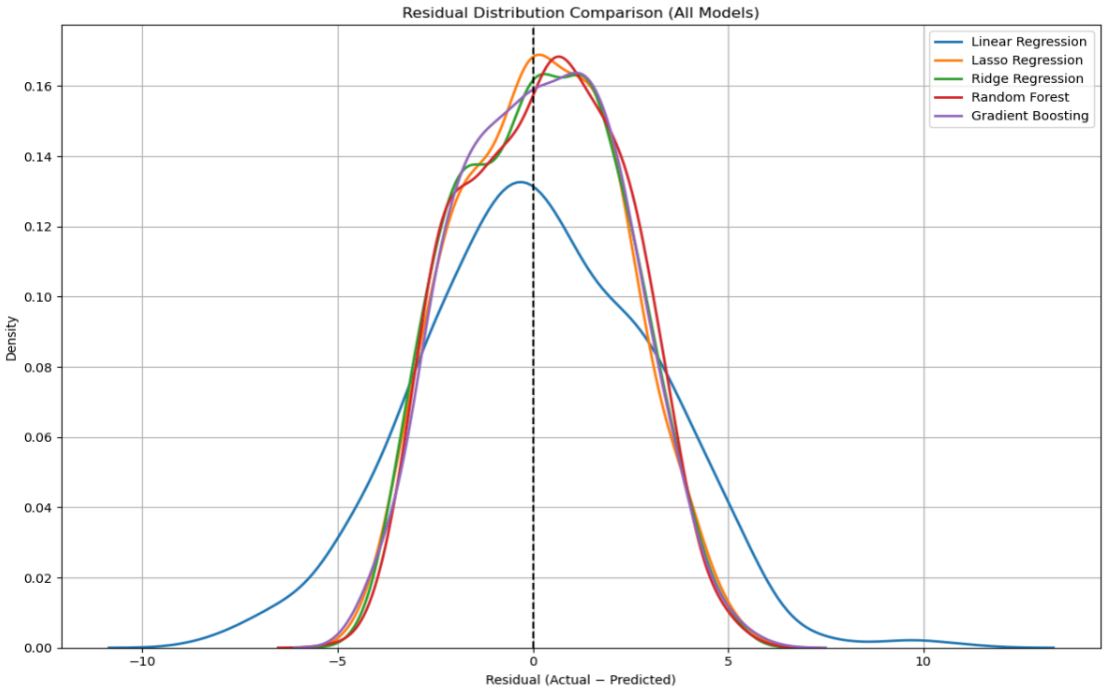
Model	MAE	RMSE
Random Forest Regressor	1.714583	2.036429
Ridge Regression	1.711827	2.044253
Lasso Regression	1.697490	2.047180
Gradient Boosting Regressor	1.707647	2.047556
Linear Regression	2.367514	2.964912

The **Random Forest Regressor** model demonstrated the lowest Root Mean Square Error (RMSE) and the second-lowest Mean Absolute Error (MAE), indicating superior performance in predicting the numerical difficulty score. This model was selected as the final regression predictor and saved as "best_regression_model.pkl".

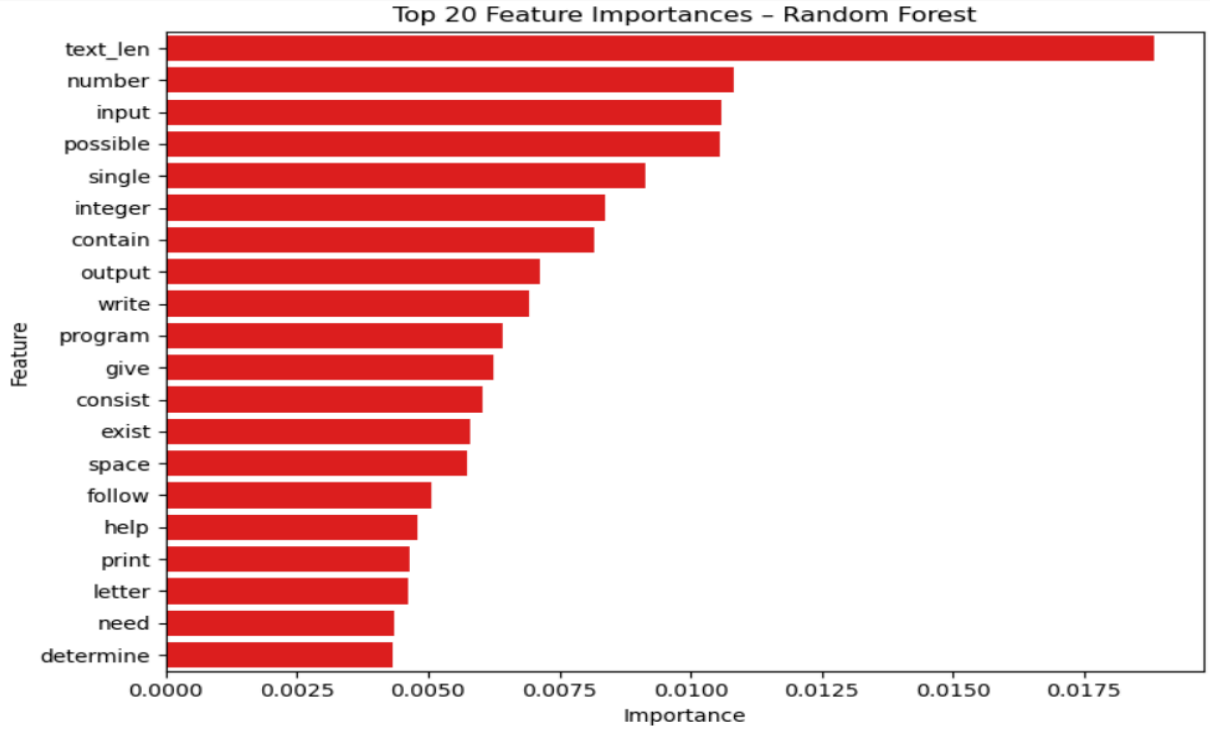
Actual vs Predicted Values (All regression models)



Residual Distribution Comparison (All regression models)



Random Forest Feature Importances

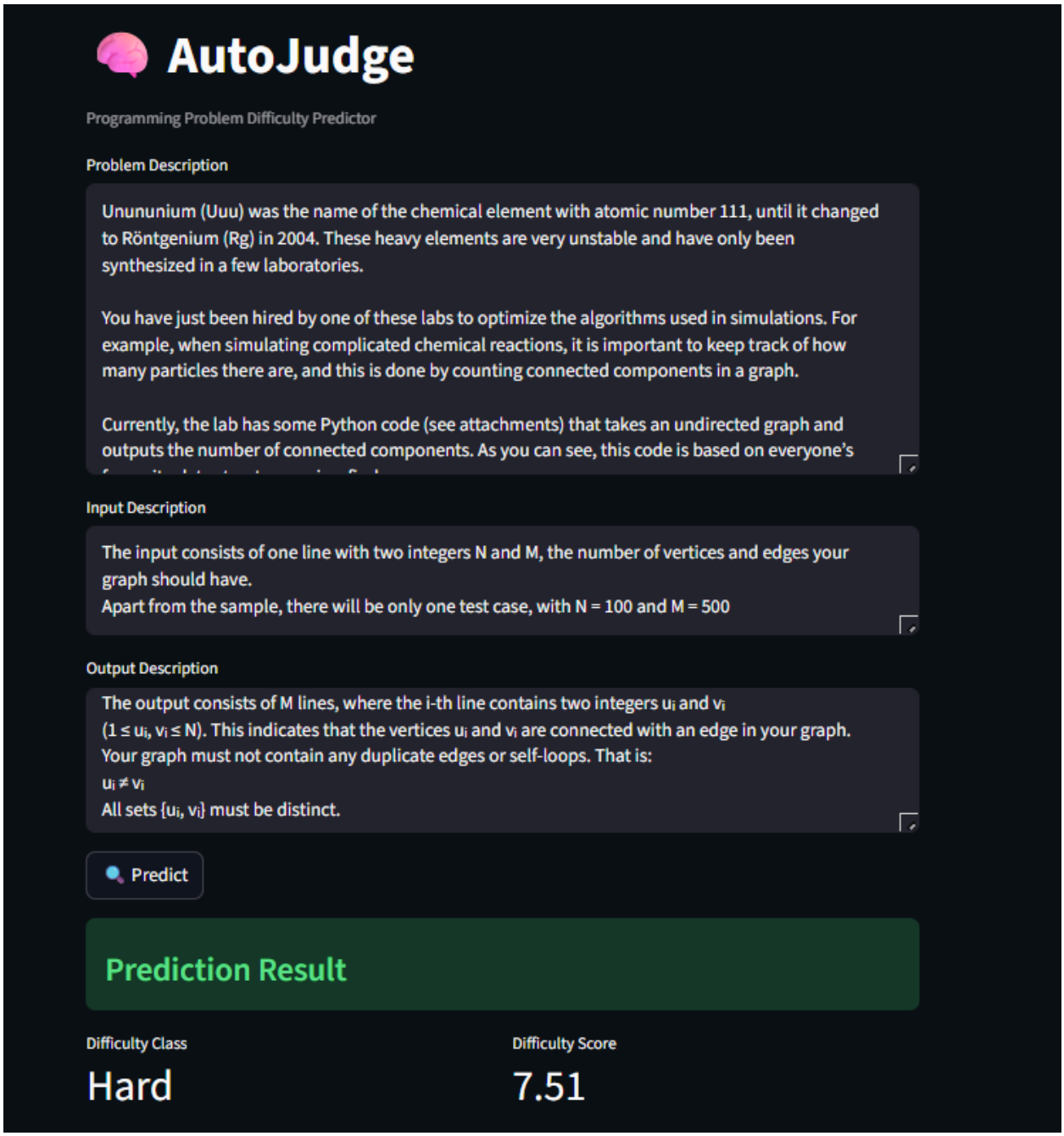


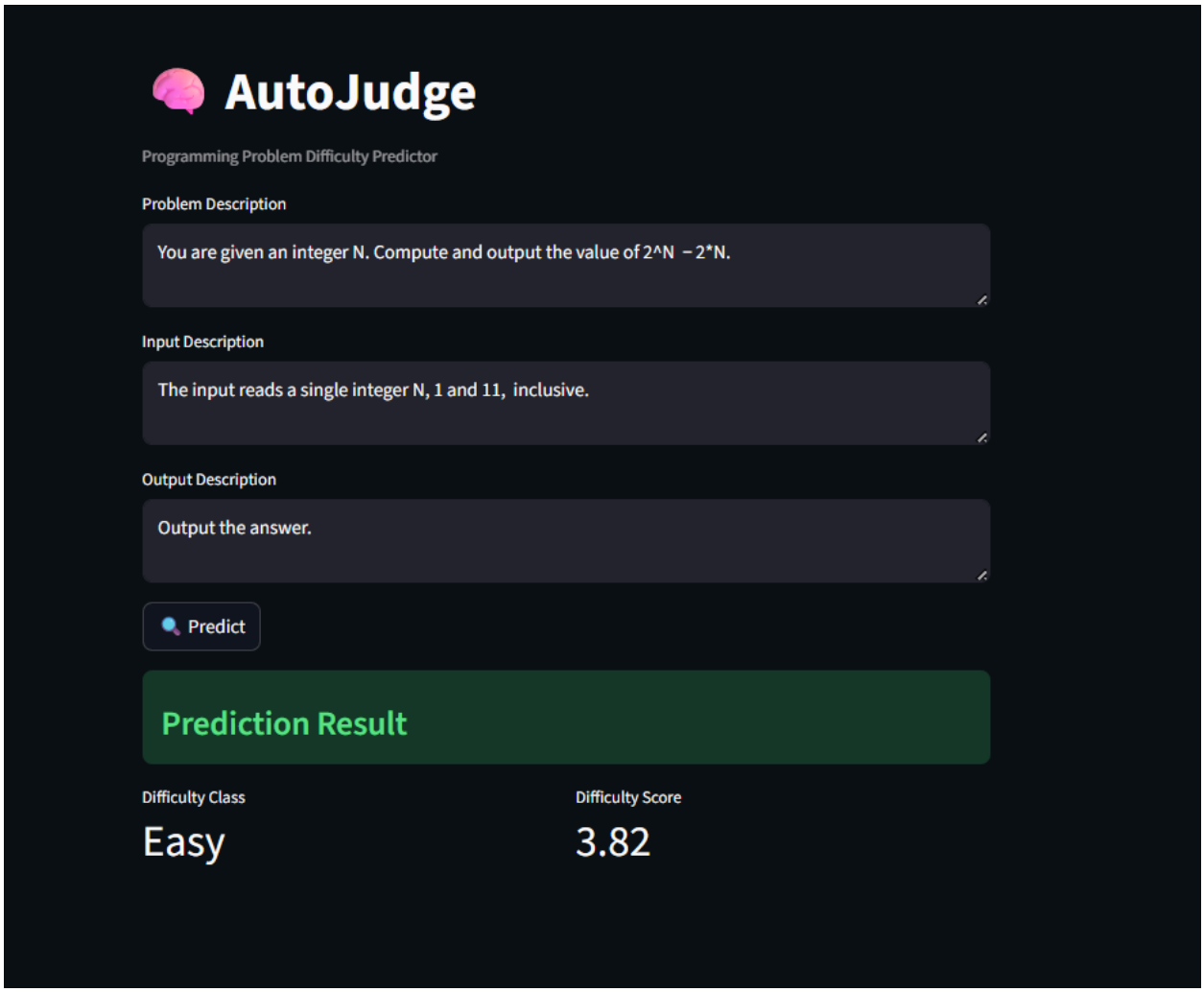
Web Interface (Streamlit)

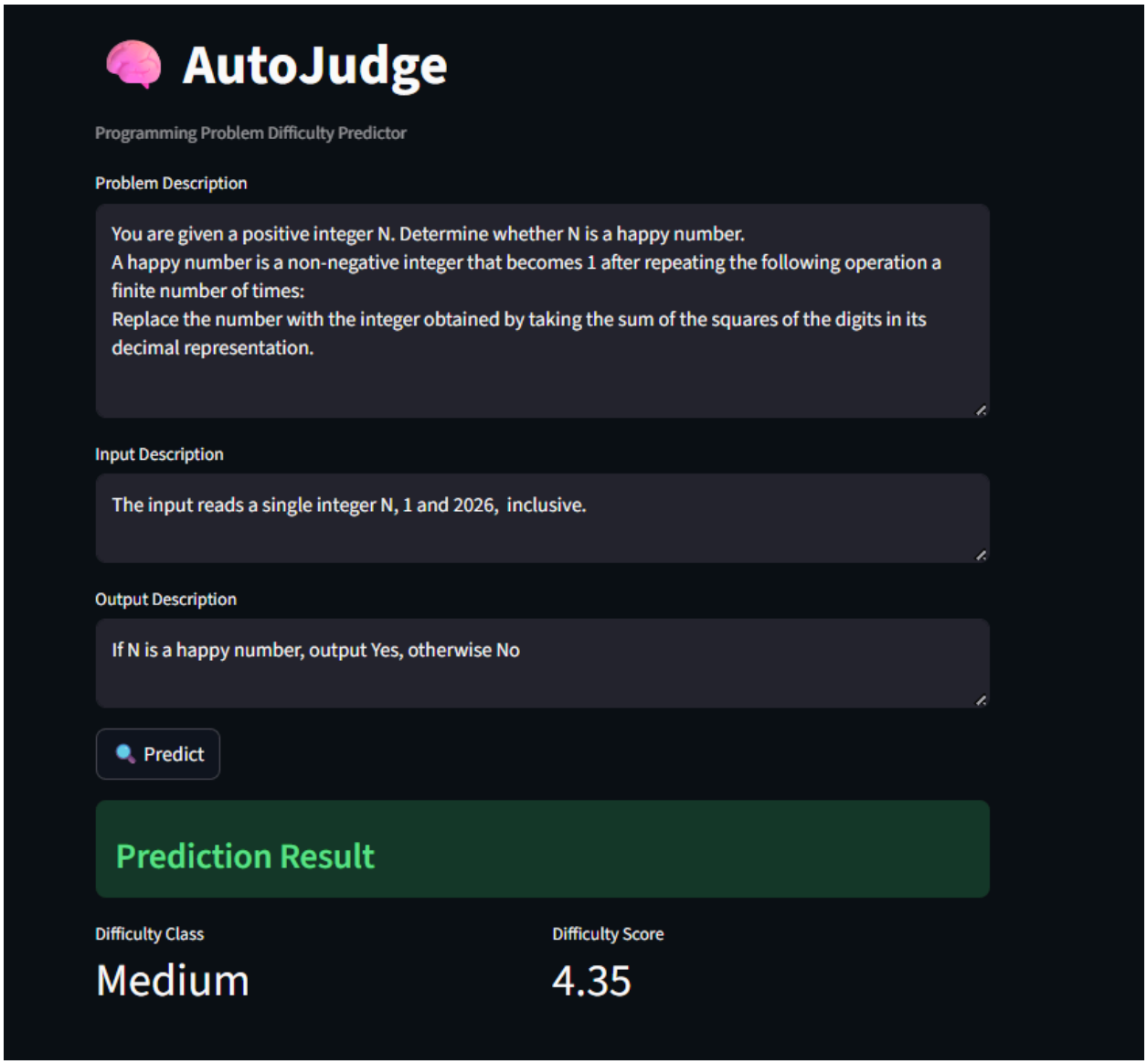
A lightweight Streamlit-based web interface is provided for interactive usage.

- Text boxes for:
    - Problem description
    - Input description
    - Output description
- One-click prediction
- Displays:
    - Predicted difficulty class
    - Predicted difficulty score

The web interface does not re-implement preprocessing. Instead, it passes user input directly to the trained pipelines, ensuring full consistency with training. Below are some sample predictions:


AutoJudge
Programming Problem Difficulty Predictor
Problem Description
Unununium (Uuu) was the name of the chemical element with atomic number 111, until it changed to Röntgenium (Rg) in 2004. These heavy elements are very unstable and have only been synthesized in a few laboratories.
You have just been hired by one of these labs to optimize the algorithms used in simulations. For example, when simulating complicated chemical reactions, it is important to keep track of how many particles there are, and this is done by counting connected components in a graph.
Currently, the lab has some Python code (see attachments) that takes an undirected graph and outputs the number of connected components. As you can see, this code is based on everyone's
Input Description
The input consists of one line with two integers N and M, the number of vertices and edges your graph should have.
Apart from the sample, there will be only one test case, with N = 100 and M = 500
Output Description
The output consists of M lines, where the i-th line contains two integers u_i and v_i
($1 \le u_i, v_i \le N$). This indicates that the vertices u_i and v_i are connected with an edge in your graph.
Your graph must not contain any duplicate edges or self-loops. That is:
$u_i \neq v_i$
All sets $\{u_i, v_i\}$ must be distinct.
Predict
Prediction Result
Difficulty Class
Hard
Difficulty Score
7.51


AutoJudge
Programming Problem Difficulty Predictor
Problem Description
You are given an integer N. Compute and output the value of 2^N −2*N.
Input Description
The input reads a single integer N, 1 and 11, inclusive.
Output Description
Output the answer.
Predict
Prediction Result
Difficulty Class
Easy
Difficulty Score
3.82


AutoJudge
Programming Problem Difficulty Predictor
Problem Description
You are given a positive integer N. Determine whether N is a happy number.
A happy number is a non-negative integer that becomes 1 after repeating the following operation a finite number of times:
Replace the number with the integer obtained by taking the sum of the squares of the digits in its decimal representation.
Input Description
The input reads a single integer N, 1 and 2026, inclusive.
Output Description
If N is a happy number, output Yes, otherwise No
Predict
Prediction Result
Difficulty Class
Medium
Difficulty Score
4.35

Conclusion

This project introduced **AutoJudge**, a **machine learning–based system** for automatically **predicting the difficulty of programming problems** using only **textual information**. By framing the task as both a **classification problem** (**Easy, Medium, Hard**) and a **regression problem** (**numerical difficulty score**), the system **eliminates reliance on user statistics or historical submission data**. A unified preprocessing pipeline combining **TF-IDF features**, **weighted keyword indicators**, and **structural features** such as text length and mathematical symbol counts ensured consistent and reliable feature extraction during both training and deployment.

Experimental evaluation showed that **Random Forest models performed best** for both classification and regression tasks, achieving the **highest accuracy** and **lowest RMSE** respectively. The integration of the trained pipelines into a **Streamlit-based web interface** demonstrates the **practical applicability** of the approach, allowing **real-time difficulty prediction** from user-provided problem descriptions. Overall, the results indicate that **textual characteristics alone** can provide meaningful signals for estimating problem difficulty, making AutoJudge a **useful baseline system** for automated problem evaluation.